Optimization

*Louden:* Finish Textbook (Chapters 1-8)

**Basic Code Generation**
- Produces functional but poor code.

**Goal:** Improve the code as much as possible.
- Dramatically improves code performance (e.g., 2X to 10X)

“*Optimization*” -- more likely “*Improvement*”

**Machine-Independent v. Machine-Dependent Optimizations**

**Variety of techniques**
- Add as many optimization algorithms as possible
  - Some are VERY complex!
- Do testing w/ sample programs to evaluate
  - which optimization strategies work best.

**Different needs for different languages** (FORTRAN)

---

**Requirement: Correctness**

Every optimization must be “safe”
- Must not change the program output ... for any input.
- Must not allow new errors or exceptions.

**Goals of optimization:**
- Runtime Execution Speed!!!
- Other (e.g., Code Size, Power Consumption)

Every optimization should improve the program
- but may slow some programs!

**Is optimization worth the effort?**
- Some algorithms may be difficult to implement.
- Many programs run only once
  - Compiler used heavily during debugging.
  - Program is only run once or twice before being modified.
  - Compiler performance matters more.

- But some programs are **computation-intensive**
  - More computation per time unit means more accurate results
The secret to getting programs to run faster?

Use a better algorithm!
The secret to getting programs to run faster? 
Use a better algorithm!

Most optimizations done by a compiler are 
“constant-factor” speed-ups 
(e.g., 25% faster)

Optimizations by the programmer:

• Change the algorithm
  \[ N^2 \rightarrow N \log N \]

• Profile the program and tweak the algorithm

• Misc. transformations

Machine Independent Optimizations

• Live Variable Analysis
• Common sub-expressions
• Eliminate unnecessary copying
• Loop transformations
  ...etc...

Optimization transforms IR Code

Machine Dependent Optimizations

• Effective Register Usage
• Select Best Target Instructions
• Select a schedule that executes quickly
  ... given the CPU idiosynchrazies
  (e.g., memory latencies, functional units, etc.)

Optimization transforms Target Code
Does the programmer trust the compiler to emit efficient code?

**No:**
Programmer will *mangle* the program to achieve greater efficiency.

**Yes:**
Programmer will concentrate on writing
- Clean, simple code
- Correct code
- Code that is easy to maintain

Is Optimization Necessary...

...assuming the programmer writes good, efficient code?

**Source Code:**

\[
A[i] := B[i] + C[i];
\]

**Translation:**

\[
\begin{align*}
t1 & := i * 4 \\
t2 & := B[t1] \\
t3 & := i * 4 \\
t4 & := C[t3] \\
t5 & := t2 + t4 \\
t6 & := i * 4 \\
A[t6] & := t5
\end{align*}
\]

The compiler will insert many hidden operations (often concerning pointers and address calculations)
“Local Transformations”
Within a single basic block

“Global Transformations”
Concern several basic blocks
(but typically within a single routine / control flow graph)

Local Common Sub-Expression Elimination

\[
\begin{align*}
B_5 :& \quad t_6 := 4 \times i \\
& \quad x := A[t_6] \\
& \quad t_7 := 4 \times i \\
& \quad t_8 := 4 \times j \\
& \quad t_9 := A[t_8] \\
& \quad A[t_7] := t_9 \\
& \quad t_{10} := 4 \times j \\
& \quad A[t_{10}] := x \\
& \quad \text{goto } B_2
\end{align*}
\]

procedure quicksort (m, n: int) is
  var i, j, v, x: int := 0;
  if (n ≤ m) then return; end;
  i := m - 1;
  j := n;
  v := A[n];
  while true do
    repeat
      i := i + 1;
      until A[i] ≥ v;
    repeat
      j := j - 1;
      until A[j] ≤ v;
      if i ≥ j then exit; end;
    x := A[i];
    A[i] := A[j];
    A[j] := x;
  end;
  x := A[i];
  A[i] := A[n];
  A[n] := x;
  quicksort (m, j);
  quicksort (i+1, n);
endProc;
procedure quicksort (m, n: int) is
  var i, j, v, x: int := 0;
  if (n ≤ m) then return; end;
  i := m - 1;
  j := n;
  v := A[n];
  while true do
    repeat
      i := i + 1;
    until A[i] ≥ v;
    repeat
      j := j - 1;
    until A[j] ≤ v;
    if i ≥ j then exit; end;
    x := A[i];
    A[i] := A[j];
    A[j] := x;
  end;
  x := A[i];
  A[i] := A[n];
  A[n] := x;
  quicksort (m, j);
  quicksort (i + 1, n);
endProc;
procedure quicksort (m, n: int) is
  var i, j, v, x: int := 0;
  if (n ≤ m) then return; end;
  i := m - 1;
  j := n;
  v := A[n];
  while true do
    repeat
      i := i + 1;
      until A[i] ≥ v;
    repeat
      j := j - 1;
      until A[j] ≤ v;
      if i ≥ j then exit; end;
      x := A[i];
      A[i] := A[j];
      A[j] := x;
    end;
    x := A[i];
    A[i] := A[n];
    A[n] := x;
    quicksort (m, j);
    quicksort (i + 1, n);
  endProc;
procedure quicksort (m,n: int) is
    var i,j,v,x: int := 0;
    if (n ≤ m) then return; end;
    i := m - 1;
    j := n;
    v := A[n];
    while true do
        repeat
            i := i + 1;
            until A[i] ≥ v;
        end;
        repeat
            j := j - 1;
            until A[j] ≤ v;
        end;
        if i ≥ j then exit; end;
        x := A[i];
        A[i] := A[j];
        A[j] := x;
    end;
    x := A[i];
    A[i] := A[n];
    A[n] := x;
    quicksort (m,j);
    quicksort(i+1,n);
endProc;
“Copy Propagation”

**A “copy”**

\[
\begin{align*}
  x &:= y \\
  \ldots \\
  z &:= b + x \\
  \ldots 
\end{align*}
\]

**Any statement that uses the value of “x’**

\[
\begin{align*}
  x &:= y \\
  \ldots \\
  z &:= b + y \\
  \ldots 
\end{align*}
\]

Why perform this optimization?
“Copy Propagation”

A “copy”

Any statement that uses the value of “x’”

Why perform this optimization?

*The copy may become DEAD CODE.*

*We may delete the copy later!*

Global Common Sub-expression Elimination

An expression...

Simple computation

Computed in several places

© Harry H. Porter, 2006
Global Common Sub-expression Elimination

An expression...
Simple computation
Computed in several places

Copies will be introduced during sub-expression elimination

...but they may be DEAD CODE!
\texttt{t6 := 4 * i}
\texttt{x := A[t6]}
\texttt{t8 := 4 * j}
\texttt{t9 := A[t8]}
\texttt{A[t6] := t9}
\texttt{A[t8] := x}

\texttt{i := m - 1}
\texttt{j := n}
\texttt{t1 := 4 * n}
\texttt{v := A[t1]}

\texttt{i := i + 1}
\texttt{t2 := 4 * i}
\texttt{t3 := A[t2]}
\texttt{if t3 < v goto B}

\texttt{j := j - 1}
\texttt{t4 := 4 * j}
\texttt{t5 := A[t4]}
\texttt{if t5 > v goto B}

\texttt{if i \geq j goto B}

\texttt{t11 := 4 * i}
\texttt{x := A[t11]}
\texttt{t13 := 4 * n}
\texttt{t14 := A[t13]}
\texttt{A[t11] := t14}
\texttt{A[t13] := x}
t6 := t2  
A[t2] := t9

Optimization, Part 1

Copy Propagation

Global Common Sub-Expression Elimination

© Harry H. Porter, 2006

Global Common Sub-Expression Elimination

© Harry H. Porter, 2006
Global Common Sub-Expression Elimination

Copy Propagation
Consider This Example

degug := FALSE;
...
if (debug) then print (...) endif

Unreachable? Can this code be eliminated?
**Consider This Example**

def = FALSE;
...
if (debug) then print (...) endif

**Data Flow Analysis**

"Which computations can reach which points"

Only one DEFINITION of "debug" can reach this USE.

Must have the value "false", so okay to optimize the IF statement.

Global Common Sub-Expression Elimination.
Copy Propagation.
Live-Variable Analysis.

**Constant Folding**

"If we know the value of a variable at compile-time, we may go ahead and perform the computation."

**Dead-Code Elimination**

"Eliminate code that is unreachable."

"Eliminate code that compute DEAD variables."

**Optimizing Loops**

**The 90-10 Rule**

"90% of execution time is spent in 10% of the code."

Try to move code out of loops

**Identify Loops**

Nesting of loops

"inner loops"

"outer loops"

**GOAL:**

*Move Code “Outward”*

*Make Loops Run Faster*
Loop-Invariant Computations

If an expression is computed within a loop and it does not depend on variables that change in the loop, then move it to just before the loop! Assume MAX and MIN are not altered in the loop.

These computations are “loop-invariant”.
Loop-Invariant Computations

If an expression is computed within a loop and it does not depend on variables that change in the loop then Move it to just before the loop!

Induction Variables

Definition: Loop counters that move together (in “lock-step”) during loop execution.

Example Source:

\[
\text{loop}
\]
\[
\ldots
\]
\[
i := i + 1;
\]
\[
\ldots \ A[i]\ldots
\]
\[
\ldots
\]
\[
\text{endloop}
\]

Translation to IR:

\[
\ldots
\]
\[
i := i + 1
\]
\[
t := i \times 4
\]
\[
\ldots \ A[t]\ldots
\]
\[
\ldots
\]

Note: t := i*4 Establishes the relationship.

Also: We know this relationship exists directly after t := i*4 is executed.
Assume $t = i \cdot 4$ here.

Some Reasoning

Then, after “$i$” is incremented, the relationship will be:

$t = (i-1) \cdot 4$

$i := i + 1$

$t := i \cdot 4$
Some Reasoning

Assume \( t = i \times 4 \) here. Then, after “\( i \)” is incremented, the relationship will be:

\[
\begin{align*}
t & = (i-1) \times 4 \\
\text{Rewriting:} & \\
t & = i \times 4 - 4 \\
\text{Or:} & \\
i & = (t + 4) / 4
\end{align*}
\]

Rewriting:

\[
\begin{align*}
t & = i \times 4 \\
i & = (t + 4) / 4
\end{align*}
\]

Use this value of “\( i \)” to compute the new “\( t \)” as a function of the old “\( t \)”.

\[
\begin{align*}
t & := i \times 4 \\
t & := [ (t + 4) / 4 ] \times 4
\end{align*}
\]
Assume \( t = i \times 4 \) here.

Then, after “i” is incremented, the relationship will be

\[
t = (i-1) \times 4
\]

Rewriting:

\[
t = i \times 4 - 4
\]

Or:

\[
i = (t + 4) / 4
\]

Use this value of “i” to compute the new “t” as a function of the old “t”.

\[
t := i \times 4
\]

\[
t := [(t + 4) / 4] \times 4
\]

Rewriting:

\[
t := t + 4
\]

**Conclusion:**

It is okay to replace \( t := i \times 4 \) by: \( t := t + 4 \)
Assume $t = i*4$ here.

Then, after “$i$” is incremented, the relationship will be

$$t = (i-1) * 4$$

Rewriting:

$$t = i*4 - 4$$

Or:

$$i = (t + 4) / 4$$

Use this value of “$i$” to compute the new “$t$” as a function of the old “$t$”.

$t := i * 4$

Rewriting:

$t := t + 4$

**Conclusion:**

It is okay to replace $t := i * 4$

by:

$t := t + 4$

*But don’t forget to establish $t = i*4$ before the loop begins!*
The Transformation

Before:

\[
\begin{align*}
\vdots \\
i &:= i + 1 \\
t &:= i \times 4 \\
\ldots \ A[t] \ldots \\
\vdots
\end{align*}
\]

Must also check:
“t” and “i” are not changed elsewhere in the loop

After:

\[
\begin{align*}
\vdots \\
t &:= i \times 4 \\
\vdots
\end{align*}
\]

“Preheader”
A new block added to “just before” the loop.

Benefit:
The definition of i may become DEAD.
... Eliminate it altogether!

© Harry H. Porter, 2006
Definitions

“Reduction in Strength”
A costly operation is replaced by a cheaper operation.
\[
t := i \times 4
\]
\[
t := t + 4
\]

“Constant Folding”
If all operands to an operator are constants... 
evaluate the operator at compile-time.
\[
t := 100 \times 4
\]
\[
t := 400
\]

Our Example, so far...

© Harry H. Porter, 2006
\[ i := m - 1 \]
\[ j := n \]
\[ t_1 := 4 \times n \]
\[ v := A[t_1] \]

\[ t_2 := 4 \times i \]
\[ t_3 := A[t_2] \]
\[ t_5 := A[t_4] \]
\[ t_4 := 4 \times j \]
\[ t_6 := A[t_6] \]

\[ \text{Induction Variables} \]

\[ j := j - 1 \]
\[ t_4 := t_4 - 4 \]
\[ t_5 := A[t_4] \]
\[ \text{if } t_5 > v \text{ goto } B_3 \]

\[ i := i + 1 \]
\[ t_2 := t_2 + 4 \]
\[ t_3 := A[t_2] \]
\[ \text{if } t_3 < v \text{ goto } B_2 \]

\[ \text{if } i \neq j \text{ goto } B_6 \]

\[ t_{14} := A[t_{14}] \]
\[ A[t_2] := t_{14} \]
\[ A[t_1] := t_3 \]
\begin{align*}
i &:= m - 1 \\
j &:= n \\
t_1 &:= 4 \times n \\
v &:= A[t_1] \\
t_2 &:= 4 \times i \\
t_4 &:= 4 \times j
\end{align*}

\textbf{Dead Code Elimination}

\begin{align*}
B_1 &:
\begin{align*}
i &:= m - 1 \\
j &:= n \\
t_1 &:= 4 \times n \\
v &:= A[t_1]
\end{align*} \\
B_2 &:
\begin{align*}
t_2 &:= t_2 + 4 \\
t_3 &:= A[t_2] \\
\text{if } t_3 &< v \text{ goto } B_2
\end{align*} \\
B_3 &:
\begin{align*}
t_4 &:= t_4 - 4 \\
t_5 &:= A[t_4] \\
\text{if } t_5 &> v \text{ goto } B_3
\end{align*} \\
B_4 &:
\begin{align*}
\text{if } t_2 \neq t_4 &\text{ goto } B_6
\end{align*} \\
B_5 &:
\begin{align*}
A[t_2] &:= t_5 \\
A[t_4] &:= t_3
\end{align*} \\
B_6 &:
\begin{align*}
t_{14} &:= A[t_1] \\
A[t_2] &:= t_{14} \\
A[t_1] &:= t_3
\end{align*}
Where are the Loops?

**Given:**
A Control Flow Graph
(Each node is a Basic Block)

**Goal:**
Locate the loops.

“Dominates”

**Definition:**
A relation between nodes...

Node D “dominates” node N

\[ D \text{ dom } N \]

If every path from the initial node to N must go through D.

The entry node in a loop will dominate all nodes in the loop.

Every node dominates itself.
“Dominates”

<table>
<thead>
<tr>
<th>Node</th>
<th>Dominates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

© Harry H. Porter, 2006
“Dominates”

<table>
<thead>
<tr>
<th>Node</th>
<th>Dominates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3 - 10</td>
</tr>
<tr>
<td>4</td>
<td>4 - 10</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

© Harry H. Porter, 2006
**Dominates**

<table>
<thead>
<tr>
<th>Node</th>
<th>Dominates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3 - 10</td>
</tr>
<tr>
<td>4</td>
<td>4 - 10</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
“Dominates”

<table>
<thead>
<tr>
<th>Node</th>
<th>Dominates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3 - 10</td>
</tr>
<tr>
<td>4</td>
<td>4 - 10</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7, 8, 9, 10</td>
</tr>
<tr>
<td>8</td>
<td>8, 9, 10</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

© Harry H. Porter, 2006
The Definition of “Natural Loops”

What is a loop anyway?

• Must have a single entry point

The Header Node

(The Header dominates all nodes in the loop.)

• Must be a path back to the header.

NOTE: This tree is different than the Control Flow Graph

Initial node will be the “root” of the dominator tree

<table>
<thead>
<tr>
<th>Node</th>
<th>Dominates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3 - 10</td>
</tr>
<tr>
<td>4</td>
<td>4 - 10</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7, 8, 9, 10</td>
</tr>
<tr>
<td>8</td>
<td>8, 9, 10</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
The Definition of “Natural Loops”

What is a loop anyway?

• Must have a single entry point
  The Header Node
  (The Header dominates all nodes in the loop.)
• Must be a path back to the header.

A loop is defined by an edge $B \rightarrow A$ such that $A$ dom $B$. 
The Definition of “Natural Loops”

What is a loop anyway?
• Must have a single entry point

The Header Node
(The Header dominates all nodes in the loop.)
• Must be a path back to the header.

A loop is defined by an edge $B \rightarrow A$ such that $A \text{ dom } B$.

Definition: Given such an edge $B \rightarrow A$, a “natural loop” is the set of nodes...
• Node A, and
• All nodes that can reach B without going through A.
Where are the Loops?

© Harry H. Porter, 2006
Where are the Loops?

while
  •
  •
  •
  if xxx   then     •     •     •
  else     •     •     •
endif
endwhile
Where are the Loops?

© Harry H. Porter, 2006
An Algorithm to Find a Natural Loop

Input: A Control Flow Graph
       A Back-Edge, B → A
Output: Result = Set of nodes in the natural loop

Stack := empty
ResultSet := \{A\}
Insert (B)
while NotEmpty (Stack) do
    M := Pop (Stack)
    for each predecessor of P of M do
        Insert (P)
    endfor
endwhile

procedure Insert(X)
    if X is not in ResultSet then
        Add X to ResultSet
        Push X onto Stack
    endif

Inner / Outer Loops

A loop is a set of nodes.

Given two Natural Loops...

Either...

• The loops are disjoint, or
• One loop is contained in (i.e., nested) within the other, or
• Both loops have the same header.

If two loops have the same header...
They will be the same loop (same set of nodes)
**Loops with Multiple Back-Edges**

Which path is traversed most frequently?
*Undecideable...*

Must treat as equally probable.

```plaintext
while (...) do
  ...A...
  ...B...
  if ... then
    ...C...
  else
    ...D...
  endif
endwhile
```

**Loop “Preheader”**

We can place loop-invariant computations in the preheader.
Reducible Control Flow Graphs

**Definition:**
In a reducible control flow graph, all loops have a single entry point.

**Structured programming constructs**
- ⇒ The control flow graph is reducible.
- ⇒ All loops are natural.

In a reducible flow graph...
We have only...
- **Forward Edges**
  These form an acyclic graph.
  All nodes can be reached via forward edges from initial node.
- **Back Edges**
  The HEAD dominates the TAIL
- **No “Cross Edges”**